CORE
Metadata, citation and similar papers at core.ac.uk
Provided by DSpace at VSB Technical University of Ostrava

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Využití paralelismu v R-stromech

Utilization of parallelism in R-tree

Bakalářská práce

2013

Jiří Kačírek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student:	**Jiří Kačírek**
Studijní program:	B2647 Informační a komunikační technologie
Studijní obor:	2612R025 Informatika a výpočetní technika
Téma:	Využití paralelizmu v R-stromech Utilization of parallelism in R-tree

Zásady pro vypracování:

Vícerozměrné datové struktury jsou relativně mladé, ale významné datové struktury. Cílem této práce je nastudovat možnosti paralelizace R-stromu – vícerozměrné datové struktury, která zůstává od jejího prvního představení nejpopulárnější dataovou strukturou pro indexování prostorových dat.

Úkoly:
1. Nastudovat si problematiku vícerozměrných datových struktur, zejména R-stromu.
2. Nastudovat si problematiku paralelního zpracování transakcí a uzamykacích protokolů pro R-stromy.
3. Začlenení paralelního přístupu do stávající implementace.
4. Porovnání výsledků se sekvenčním zpracovaním dotazů a s komerčními SŘBD.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Peter Chovanec**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Dne: *6. května 2013*

.....

podpis studenta

Za vypracování této práce vděčím především několika ochotným lidem z prostředí internetu, kteří sdílejí své vědomosti s ostatními. Dále pak mé škole, která mi dala dobrý základ a vedoucímu mé bakalářské práce, který mi poskytl odbornou pomoc.

Abstrakt

Úkolem této bakalářské práce je studie popisující princip R-stromů a jejich paralelizace včetně následné praktické realizace, v rámci které je změřena rychlost paralelismu a jsou porovnány různé způsoby jeho implementace.

Klíčová slova

R-strom, paralelismus, uzamykání, synchronizační mechanismy

Abstract

The goals of this Bachelor's dissertation is study of the R-tree and their parallelism including their practical realization in which performance of this paralellelism has been measured and compared with various possible implementations.

Klíčová slov

R-tree, parallelism, locking, synchronization mechanism

Seznam použitých symbolů a zkratek

SŘBD	Systém řízení báze dat
QuickDB	Implementace SŘBD vytvořená na katedře informatiky VŠB-TUO FEI
API	Zkratka pro Application Programming Interface, označující v informatice rozhraní pro programování aplikací
R-strom	Stromová datová struktura
Cache	Označení pro vyrovnávací paměť používanou ve výpočetní technice. Je zařazena mezi dva subsystémy s různou rychlostí a vyrovnává tak rychlost přístupu k datům
Boost	Název souboru knihoven pro jazyk C++
Mutex	Vzájemné vyloučení (anglicky mutual exclusion, nebo zkráceně mutex) je algoritmus používaný v programování jako synchronizační mechanismus
OOP	Objektově orientovaný přístup

Obsah

Seznam tabulek

Seznam ilustrací

1 Úvod

Již několik let se v osobních počítačích běžně používají více jádrové procesory. Chceme-li naprogramovat aplikaci tak, aby výpočetní kapacitu více jádrového procesoru naplno využila, je potřeba její algoritmus vytvořit sofistikovaněji než je tomu u běžných sekvenčně pracujících algoritmů.

Programování paralelně běžící aplikace vyžaduje správné zadání práce jednotlivým procesům a řízení jejich souběhu tak, aby nedocházelo ke konfliktům, a pokud k nim dojde, zajistit jejich správné řešení. Můžeme vytvořit základní funkční souběh procesů, avšak optimalizování takového programu tak, aby byl pokud možno co nejrychlejší, vyžaduje větší časové úsilí. Existuje více cest, jak vytvořit dobrou logiku paralelismu v naší aplikaci.

Cílem této práce je implementovat paralelismus do frameworku QuickDB vyvíjeného na katedře informatiky VŠB-TUO FEI. Ten obsahuje perzistentní datové struktury, mezi kterými je i R-strom, kterého se bude týkat tato bakalářské práce.

Ve druhé kapitole tohoto dokumentu se nachází teoretický nástin řešené problematiky. Obsahuje popis toho, co jsou R-stromy, vysvětlení pojmu transakce, principy uzamykacích protokolů pro transakce a jejich souvislost s R-stromy. Dále je popsána problematika paralelismu, jsou vysvětleny pojmy vlákna a řízení jejich souběhu a ukázány různé modely pro vytváření paralelismu. Ve třetí kapitole se nachází popis způsobu implementace nutné k realizaci této práce. Jsou zde popsány způsoby práce se synchronizačními mechanismy, které byly použity k řízení souběhu vláken. Ve čtvrté kapitole jsou ukázány výsledky testování (měření) paralelismu a jeho porovnání se sekvenčním přístupem. Poslední pátá kapitola je shrnutím dosažených výsledků.

2 Teoretické vysvětlení problematiky

2.1 R-stromy

R-stromy[3] byly poprvé představeny roku 1984 Antonínem Guttmanem jako modifikace B-stromů pro indexování prostorových dat.

R-strom je tvořen vnitřními a listovými uzly. V listových uzlech jsou uloženy listové položky obsahující ukazatele na datové objekty reprezentující prostorové objekty.

Jde o dynamickou strukturu, která má polohu svých dat určenou tzv. *minimálním ohraničujícím hyperkvádrem (minimall bounding box)* - MBB. *V* dvourozměrném prostoru se bude jednat o obdélník, ve trojrozměrném o kvádr atd. MBB je určen dvěma body, Q_L a Q_H.

U R-stromu nemusí být před vyhledáváním provedena reorganizace indexu ani periodická reorganizace. Vnitřní uzel U obsahuje dvojici (I_i, P_i), kde P_i je ukazatel na potomka uzlu *U*, tj. na uzel C_i, I_i je MBB uzlu U_i a platí, že pokud je I MBB uzlu *U,* pak všechny I_i potomků uzlu U jsou v *I* obsaženy.

Listová položka v listu R-stromu je dvojice (I_i, O_i), kde O_i je ukazatel na položku v databázi obsahující daný prostorový objekt a I_i je MBB objektu.

Pro *M* maximální počet položek, které vyplní jeden uzel, a parametr *m, m <= M/2,* určující minimální počet položek v uzlu R-stromu, platí následující:

- Každý listový uzel obsahuje *m* až *M* indexových položek
- Každý vnitřní uzel má *m* až *M* potomků
- Nelistový kořen musí mít nejméně dva potomky
- Všechny listy jsou ve stromu na stejné úrovni - je výškově vyvážený

Maximální výška H_{max} a maximální počet uzlů C_{max} *m*-árního R-stromu s *N* indexovými záznamy jsou:

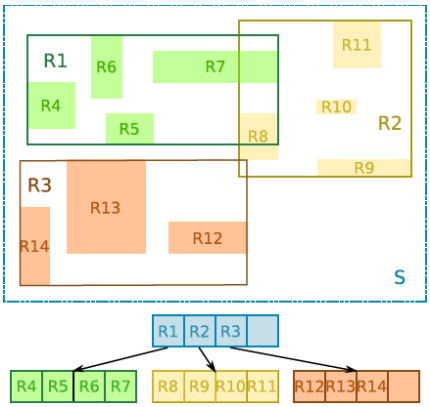
$$H_{max} = [\log_m N] - 1$$

$$C_{max} = \left[\frac{N}{m}\right] + \left[\frac{N}{m^2}\right] + \ldots + 1$$

Nejhorší případ naplnění uzlu (utilization) λ nekořenových uzlů je:

$$\lambda = \frac{m}{M} \quad \text{resp.} \quad \lambda = \frac{m}{M} \cdot 100[\%]$$

Při dodržení pravidla minimálně polovičního zaplnění uzlu nebude nikdy $\lambda < 0{,}5$ resp. $\lambda < 50\%$.



Ilustrace 1: R-strom

Výkon operací vyhledávání a vkládání (a s ním spojené štěpení uzlu) položek v R-stromech určuje míra *pokrytí* úrovně stromu, což je celková plocha dané úrovně stromu, která ovlivňuje velikost mrtvé plochy a prohledávaného prostoru. [4]

Vyhledávání

Je dán R-strom, jeho kořen U a vyhledávací hyperkvádr *QHB* (tvořen dvěma body Q_L a Q_H) . Úkolem je najít všechny indexové záznamy, jejichž MBB se překrývají s *QHB*. Vyhledávací algoritmus tvoří dvě fáze: v první se prohledávají podstromy, ve druhé listové uzly. V obou fázích se pak u každé položky *E* vnitřního resp. listového uzlu U ověří, zda se MBB I této položky překrývá s QHB. Pokud ano, v případě vnitřního uzlu se nad všemi takovými položkami rekurzivně zavolá vyhledávání, přičemž za kořen *U* bude tentokrát dosazen uzel, na nějž je příslušnou položkou odkazováno. V případě listového uzlu pak položka *E*, která se s QHB překrývá, splňuje úkol a bude jednou z navrácených položek.

Vyhledávání je komplikováno faktem, že jednotlivé MBB se mohou překrývat. Vezmeme si například R-strom, jak je znázorněno na obrázku výše. Kdyby hledaný objekt ležel v průniku R3 a R4, bylo by nutné prohledat oba podstromy, které pod těmito uzly leží.

Vkládání

Při vkládání objektu do stromu rozhoduje minimalizace pokrytí úrovně stromu, čímž se minimalizuje mrtvá plocha a přesah na úrovni stromu, což vede ke snížení počtu prohledávaných podstromů.

Nejdříve je třeba najít vhodný listový uzel, do nějž bude indexový záznam vložen. Začíná se od kořene a vždy se vybere potomek, který, pokud do něj bude záznam vložen, potřebuje nejmenší rozšíření. Pokud je více kandidátů, vybere se ten uzel, jehož výsledná plocha bude nejmenší. Tímto způsobem se pak rekurzivně pokračuje až k listu, do nějž je indexový záznam vložen, ovšem pouze v případě, že je *bezpečný,*tj. že přidáním položky nedojde k přetečení uzlu. Pokud není *bezpečný*, musí být tento (listový) uzel rozdělen na dva (operací *split*) a jeho položky

přerozděleny. Rozdělením listového uzlu na dva vznikne v rodičovském uzlu nová položka, čímž může opět dojít k přetečení a opětovné a opětovné reorganizaci. Toto přetékání se může šířit dále až ke kořeni. Pokud dojde k přetečení i v kořeni, musí dojít k jeho štěpení, přerozdělení jeho položek a zavedení nového kořene (výška stromu se tak zvýší).

2.2 Uzamykací protokoly

2.2.1 Definice transakcí

Transakce je základní jednotkou práce s databází. Je to posloupnost akcí, jako jsou vkládání, mazání a aktualizace položek v databázi. Každá transakce musí splňovat tzv. vlastnost ACID[5] - Atomicity (atomičnost), Consistency (konzistence), Isolation (izolovanost), Durability (trvalost).

- Atomicity - transakce je jednotka - ačkoliv se skládá z jednotlivých akcí, musí proběhnout celá nebo vůbec ne. Transakce má na starost komponenta zotavení z chyb. Musí zajistit, aby v případě chyby při vykonávání transakce byla databáze uvedena do předchozího stavu.
- Consistency - transakce transformuje databázi z jednoho konzistentního stavu do druhého.
- Isolation - transakce jsou navzájem izolované, tj. dílčí efekty jedné transakce nejsou viditelné jinou transakcí.
- Durability - Efekty úspěšně provedené transakce jsou perzistentní (uloženy v databázi).

Pro zajištění vlastnosti ACID slouží operace COMMIT pro potvrzení úspěšně dokončené transakce a ROLLBACK pro ohlášení chyby transakce a reorganizace databáze do konzistentního stavu před zahájením transakce.

2.2.2 Paralelní zpracování transakcí

Z hlediska využití zdrojů je žádoucí, abychom mohli transakce zpracovávat paralelně. Požadavek na paralelní zpracování transakcí je takový, aby výsledek paralelního zpracování byl totožný se sériovým, tomu říkáme *uspořadatelnost*[6].

Při paralelním zpracovávání transakcí však může docházet k tzv. *anomáliím,* tj. k chybám, kvůli kterým je databáze v nekonzistentním stavu - jinými slovy výsledky transakcí jsou vlivem jejich špatného prokládání chybné a tedy zároveň odlišné od výsledků, které bychom získali při jejich sériovém zpracování. Existují tyto anomálie, které mohou vlivem špatného prokládání transakcí vzniknout:

- Ztráta aktualizace
- Neopakovatelné čtení
- Fantomové čtení

Anomálie *ztráta aktualizace* nastane v následující situaci. Máme transakce T_1 a T_2 vykonávány sériově:

T_1	T_2	Stav
READ(A)		A=1
A+=5		inkrementace A o 5
WRITE(A)		A =6
	READ(A)	A = 6
	A+=3	inkrementace A o 3
	WRITE(A)	A = 9

Tabulka 1: Příklad sériového vykonání transakcí

Jak je vidět, konečná hodnota proměnné A je 9. V následující tabulce je ukázáno prokládané provedení těchto transakcí:

T_1	T_2	Stav
READ(A)		A = 1
A+=5		inkrementace A o 5
	READ(A)	A = 1
	A+=3	inkrementace A o 3
WRITE(A)		A = 6
	WRITE(A)	A = 4

Tabulka 2: Příklad paralelního vykonání transakcí vedoucího ke ztrátě aktualizace

Transakce T_1 čte A a inkrementuje jí o hodnotu 5, avšak ještě než stačí aktualizaci potvrdit příkazem *WRITE*, čte transakce T_2 stále starou hodnotu A a provede vlastní inkrementaci. T_1 následně provede *WRITE* a T_2 pak tuto hodnotu přepíše. Tím dojde ke ztrátě aktualizace transakce T_1.

Anomálie *neopakovatelné čtení* nastane *v* situaci, kdy chce transakce T_1 přečíst hodnotu, kterou již předtím přečetla, avšak mezitím ji již změnila transakce T_2.

Anomálie *fantomové čtení* vznikne, načte-li transakce T_1 množinu hodnot, se kterými bude dále pracovat, ale transakce T_2 tuto množinu modifikuje operací Insert nebo Delete. Bude-li transakce T_1 následně načítat stejnou množinu, bude již jiná.

Řada SŘBD umožňuje nastavit tzv. *úroveň izolace* značenou čísly 0-3. Čím vyšší *úroveň izolace* je, tím je menší riziko vzniku anomálií:

Úroveň izolace	Anomálie		
	ztráta aktualizace	neopakovatelné čtení	fantomové čtení
0-nepotvrzené čtení	vzniká	vzniká	vzniká
1-potvrzené čtení	-	vzniká	vzniká
2-opakovatelné čtení	-	-	vzniká
3-uspořadatelný rozvrh	-	-	-

Tabulka 3: Úroveň izolace podle SQL92

2.2.3 Uzamykací protokoly

Návrh rozvrhu, jak budou transakce prokládány, a jeho testování ekvivalence se sériově vykonávaným rozvrhem, je ve třídě NP-úplných problémů. Proto je tento přístup pro zajištění konzistence databáze nepřijatelný. Transakce je však možné rozvrhnout podle sady pravidel, která zajišťují uspořadatelnost takového rozvrhu - pomocí *uzamykacích protokolů*.

Jsou postaveny na uzamykání položek operacemi LOCK a UNLOCK. Provádí-li transakce operace s položkou (např. čtení) uzamkne ji operací LOCK. Ostatní transakce musí čekat na odemknutí položky operací UNLOCK.

Pro zlepšení vlastnosti paralelního souběhu jednotlivých transakcí však existuje více typů uzamčení. Základní dva jsou sdílený a výlučný zámek:

- *sdílený zámek* S_LOCK na položku v databázi může držet více, než jedna transakce
- *výlučný zámek* X_LOCK *na* položku v databázi může být držen vždy pouze jednou transakcí

V případě, že bude chtít transakce T_1 pouze číst položku A, bude ji udělen zámek S_LOCK(A). Pokud transakce T_2 bude chtít zrovna také číst položku A, bude jí také udělen zámek S_LOCK(A). Kdyby však chtěla transakce T_2 položku A modifikovat, bude muset získat zámek X_LOCK(A). Ten může získat až po uvolnění zámku S_LOCK(A) držený transakcí T_1.

Používání sdílených a výlučných zámků se řídí následujícím protokolem:

1. Před provedení operace READ(X) musí transakce uzamknout objekt X sdíleným S_LOCK(X) nebo výlučným X_LOCK(X) zámkem.
2. Před provedením operace WRITE(X) musí transakce uzamknout objekt X výlučným zámkem X_LOCK.
3. Poté, co transakce provedla všechny operace READ(X) a WRITE(X), musí zámek uvolnit (Toto pravidlo nemusí být striktně dodrženo).
4. Transakce nesmí provést S_LOCK(X), pokud již objekt X uzamkla sdíleným či výlučným zámkem.

5. Transakce nesmí provést X_LOCK(X), pokud již objekt X uzamkla sdíleným či výlučným zámkem.

Při používání zámků může nastat situace uváznutí, tzv. *deadlock.* jako je to v tomto příkladě:

T_1	T_2
X_LOCK(A)	
WRITE(A)	
	X_LOCK(B)
	WRITE(B)
X_LOCK(B)	
WRITE(B)	
UNLOCK(A)	
UNLOCK(B)	
	X_LOCK(A)
	WRITE(A)
	UNLOCK(A)
	UNLOCK(B)

Tabulka 4: Vznik *deadlocku*

Transakce T_1 a T_2 na sebe navzájem čekají na uvolnění zámků X_LOCK. Tato situace se řeší zrušením jedné z transakcí dle různých kritérií, například podle množství odvedeného úsilí, které už transakce provedla apod.

Kromě výlučných a sdílených zámků existuje také strukturované zamykání. Dlouhotrvající transakce, tj. takové, které přistupují k většímu množství objektů, proto mohou zamykat celé celky, tzv. větší granule. To mohou být např. sloupce tabulky, celé tabulky, soubory atd. Kdyby měly takovéto transakce zamykat všechny položky, ke kterým budou přistupovat, režie zamykání by pak byla velmi vysoká. Jsou zde proto také tzv. *záměrné zámky (intention locks)*, které uzamykají všechny předchůdce dané granule (tj. všechny granule, které ji obsahují) podle operace, která má být nad granulí provedena. Jsou to zámky pro *záměrné čtení (intention shared lock)* - IS_LOCK, *záměrný zápis (intention exclusive lock)* - IX_LOCK a *záměrné čtení s následným zápisem (shared with exclusive lock)* - SIX_LOCK. Řídí se následujícím protokolem:

1. Zámky musejí být drženy ve shodě s maticí kompatibility (viz tabulka níže).
2. Ve kterémkoliv módu musí být jako první uzamčen kořen hierarchie.
3. Uzel hierarchie může být transakcí uzamčen pomocí S_LOCK nebo IS_LOCK, pouze pokud byl tutéž transakcí uzamčen pomocí IS_LOCK nebo IX_LOCK jeho přímý předchůdce.
4. Uzel může být transakcí uzamčen zámkem X_LOCK, IX_LOCK nebo SIX_LOCK pouze pokud byl tutéž transakcí uzamčen jeho přímý předek zámkem IX_LOCK nebo

SIX_LOCK.

5. Transakce může odemknout uzel, pouze pokud žádný z jeho potomků není uzamčen.
6. Transakce může zamknout uzel, pouze pokud předtím žádný uzel neodemykala.

	S	X	IS	IX	SIX
S	√	-	√	-	-
X	-	-	-	-	-
IS	√	-	√	√	√
IX	-	-	√	√	-
SIX	-	-	√	-	-

Tabulka 5: Matice kompatibility současně držených zámků

Pokud jde o protokoly pro zamykání R-stromů, byly pro ně navrženy *stromové uzamykací protokoly.* Budeme-li předpokládat použití pouze výlučných a sdílených zámků, je protokol následující:

1. Transakce *T* může použít první X_LOCK na jakémkoliv objektu X.

2. Další objekt může být uzamčen transakcí *T*, pouze když byl touto transakcí uzamčen jeho předchůdce.

3. Objekty mohou být odemknuty kdykoliv.

4. Objekt, který byl transakcí *T* zamknut a odemknut již nemůže být touto transakcí znovu zamknut.

Z uvedeného vyplývá, že transakce *T* může odemknout *X* pouze, pokud uzamkla všechny potomky uzlu X. Tomuto přístupu se říká *lock coupling*.

Toto byl nástin problematiky zamykání v databázích. Pro zamykání R-stromů existuje řada uzamykacích protokolů. Důležité zde je si uvědomit, že práce s R-stromem není jen o vkládání prvků a dotazování se na ně a tedy zajištění, aby se tyto operace navzájem neovlivňovaly, ale že je třeba zajistit dokonce vzájemné neovlivňování se jednotlivých transakcí, které s R-stromem pracují, tedy množiny úloh (vkládání, dotazování, mazání atd.), které navzájem nesmí kolidovat jakožto celky.

2.3 Paralelismus v počítačové aplikaci

2.3.1 Definice a případy užití vláken

Vlákno [1] (anglicky: thread) v informatice označuje vlákno výpočtu neboli samostatný výpočetní tok, tedy posloupnost po sobě jdoucích operací. Každá spuštěná aplikace má alespoň jeden proces a každý proces má alespoň jedno vlákno, ve kterém počítá. Dříve platilo, že proces měl jen jedno vlákno (přesněji, nebylo důvod tyto pojmy odlišovat). Dnes je stále více programů multithreadových (více-vláknových), tedy uvnitř jednoho procesu (a v jednom adresovém prostoru, tedy se sdílenou pamětí) může zároveň běžet více vláken.

Veškerá správa vláken je řízena OS a každé vlákno v uživatelském prostoru je zobrazeno na vlákno v jádře OS. Samotné jádro vytváří, ruší a plánuje vlákna.

Vlákna se typicky používají v těchto situacích:

- servery - je třeba obsluhovat paralelně několik klientů najednou
- výpočetní operace - u více-jádrových procesorů je možné rozdělit výpočetní operaci do několik pod-úloh a urychlit tak výpočet
- interface aplikací - po dobu zpracování požadavku (např. stisknutí tlačítka pro časově náročný výpočet) není dobré, aby interface aplikace zamrz, proto je lepší, aby běžel v samostatném vlákně

2.3.2 Modely vícevláknových aplikací

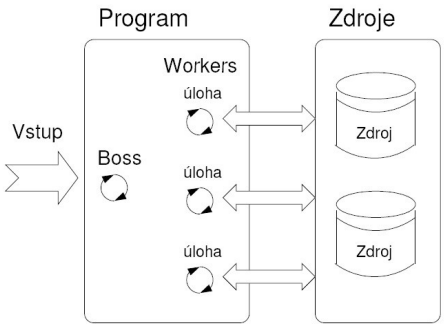
Přístupů k vytvoření paralelismu v aplikaci je více.

Modely[1] řeší způsob vytváření a rozdělování práce mezi vlákna.

- Boss/Worker - hlavní vlákno řídí rozdělení úlohy jiným vláknům
- Peer - vlákna běží paralelně bez specifického vedoucího
- Pipeline - zpracování dat sekvencí operací; předpokládá dlouhý vstupní proud dat

Model Boss/Worker

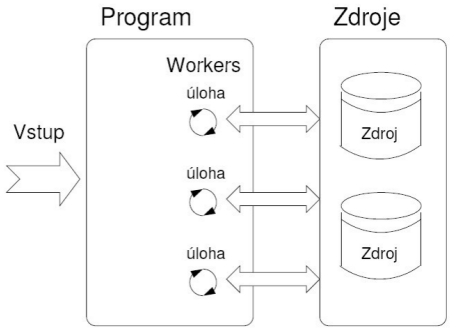
- Hlavní vlákno je zodpovědné za vyřizování požadavků. Pracuje v cyklu:
    1. Příchod požadavku
    2. Vytvoření vlákna pro řešení příslušného úkolu
    3. Návrat na čekání požadavku
- Výstup řešení úkolu je řízen:
    - příslušným vláknem řešícím úkol
    - hlavním vláknem - předání využívá synchronizační mechanismy



Ilustrace 2: Model Boss/Worker

Peer model

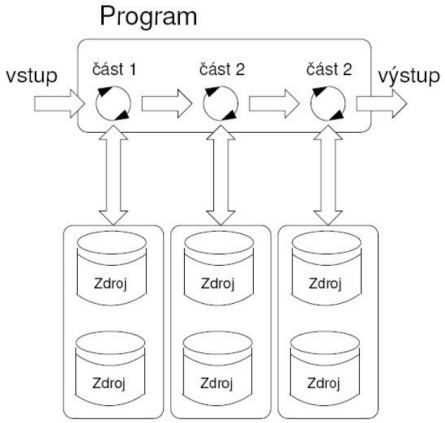
- Neobsahuje hlavní vlákno
- První vlákno po vytvoření ostatních vláken:
    - se stává jedním z ostatních vláken (rovnocenným)
    - pozastavuje svou činnost do doby, než ostatní vlákna skončí



Ilustrace 3: Model Peer

Model Pipeline

- Každé vlákno je zodpovědné za svůj vstup a výstup
- Dlouhý vstupní proud dat
- Sekvence operací (části zpracování), každá vstupní jednotka musí projít všemi částmi zpracování
- V každé části jsou v daném čase zpracovány různé jednotky vstupu (nezávislost jednotek)



Ilustrace 4: Model Pipeline

2.3.3 Řízení souběhu

Řízení souběhu je nutné, pokud může v paralelně běžící aplikaci nastat situace, kdy dvě a více vláken budou chtít přistupovat ve stejný okamžik ke stejnému zdroji, např. k proměnné uložené v paměti. Pokud obě vlákna budou chtít tuto proměnnou modifikovat ve stejný čas, dojde při běhu aplikace k chybě. Navíc, pokud je určitý zdroj modifikován jedním vláknem a čten druhým vláknem ve stejný čas, přečtená data mohou být nesprávná.

Těmto případům se dá předcházet použitím tzv. *synchronizačních mechanismů*. Slouží k tomu, aby mohlo být v kritické oblasti kódu vždy pouze jedno vlákno, eventuálně více vláken, které se nebudou vzájemně blokovat.

Synchronizační mechanismy

Synchronizační mechanismy se používají pro řízení souběhu více vláken. Pro účely řízení paralelismu v této bakalářské práci bylo použito několik typů synchronizačních mechanismů z důvodu porovnání jejich výkonosti. Všechny však v základu pracují na podobném principu.

Definujme si jako kritickou oblast[2] tu část kódu, ve kterém by mohlo dojít k chybám souběhu více vláken, např. v případě modifikace proměnné, která je sdílena mezi všemi vlákny a synchronizační mechanismus jako nástroj, který zajišťuje správný souběh vláken, které chtějí do této kritické oblasti vstoupit.

Problémem synchronizačních mechanismů je pak umožnit přístup ke kritické oblasti vláknům, které o to usilují, při dodržení následujících podmínek:

- výhradní přístup; v každém okamžiku smí být v kritické oblasti nejvýše jedno vlákno
- vývoj; rozhodování o tom, který proces vstoupí do kritické oblasti, ovlivňují pouze vlákna, které o vstup do kritické oblasti usilují; toto rozhodnutí pro žádné vlákno nemůže být odkládáno do nekonečna; nedodržení této podmínky může vést například k tomu, že je umožněna pouze striktní alternace (dva procesy se při průchodu kritickou sekcí musí pravidelně střídat)
- omezené čekání; pokud jedno vlákno usiluje o vstup do kritické oblasti, nemohou ostatní procesy tomuto vstupu zabránit tím, že se v kritické oblasti neustále střídají – mohou do této kritické oblasti vstoupit pouze omezený počet krát (zpravidla pouze jednou)

Pokud o přístup do kritické oblasti usiluje některé vlákno v době, kdy je v ní jiné vlákno, případně o přístup usiluje v jednom okamžiku více vláken, je nutné některé z nich pozdržet. Toto pozdržení je možné realizovat smyčkou. Toto tzv. *aktivní čekání* (*busy waiting*) však zbytečně spotřebovává čas CPU – je možné čekající vlákno zablokovat a obnovit jeho běh až v okamžiku, kdy vlákno, které je v kritické oblasti, tuto oblast opustí.

Je možné vytvořit svou vlastní logiku řízení vláken, například pomocí smyčky *While,* která se neustále dotazuje na uvolnění konkrétních zdrojů, tento způsob však není dostatečně výkonný ve srovnání s logikou synchronizačních mechanismů, které jsou pro tento účel vytvořené.

3 Implementace

Úkolem je začlenění paralelního přístupu do stávající implementace. Tento paralelní přístup se však týká jen dotazování na prvky R-stromu, nikoliv na vkládání. Specifikace úkolu je následující:

- spustit operace dotazování ve více vláknech
- uzamykacími mechanismy zajistit, aby při vykonávání dotazů v jednotlivých běžících vláknech nedocházelo k chybám souběhu
- použít různé uzamykací mechanismy pro porovnání jejich výkonosti

V následujících podkapitolách následuje popis jednotlivých úkonů.

3.1 Synchronizační mechanismy

Aplikace QuickDB je napsána tak, že několik tříd, které jsou sdíleny mezi jednotlivými běžícími vlákny, musí být uvnitř ošetřeno synchronizačními mechanismy. Pokud je nějaká třída ošetřena tak, aby v ní nedošlo ke kolizi, pracuje-li více vláken se stejnou instancí této třídy, říkáme o ní, že je tzv. *thread-safe* (vláknově bezpečná). Toto je zajištěno těmito synchronizačními mechanismy:

- *Kritické sekce* knihovny Windows API
- *Mutex* knihovny Windows API
- *Shared Mutex* knihovny Boost
- *Metered Section* poskytnutý vedoucím bakalářské práce

3.1.1 Kritické sekce

Kritické sekce se používají jako prvek řízení souběhu více vláken na těch místech kódu (v metodách tříd), ve kterých k souběhu více vláken může dojít.

Celá procedura použití kritické sekce vypadá následovně:

```
CRITICAL_SECTION criticalSection;                       //definice proměnné
InitializeCriticalSection(&criticalSection);            //inicializace kritické sekce
EnterCriticalSection(&criticalSection);                 //vstup do kritické sekce
/*kód ve kterém by nmohlo dojít ke kolizi více vláken*/
LeaveCriticalSection(&criticalSection);                 //výstup z kritické sekce
DeleteCriticalSection(&criticalSection);                //dealokace kritické sekce
```

Jak je vidět, do *kritické sekce se* vstupuje pomocí funkce *EnterCriticalSection()* a vystupuje se z ní pomocí funkce *LeaveCriticalSection()*. Kód, který se nachází mezi těmito funkcemi, má zajištěno, že v něm bude vždy jen jedno z *vláken*. Tento způsob však není příliš vhodný v mnoha částech již napsané implementace. Máme-li například jednoduchou metodu, která má za úkol vrátit určitý atribut třídy, je napsána většinou takto:

```
int Trida::GetValue()
{
        return value;
}
```

Pokud potřebujeme zajistit, aby byl atribut *value* třídy *Trida* chráněn proti úpravám více *vláken* najednou, musíme přidat kritickou sekci:

```
int Trida::GetValue()
{
        EnterCriticalSection(&kritickaSekceTridy);
        int ret = value;
        LeaveCriticalSection(&kritickaSekceTridy);
        return ret;
}
```

Jak je vidět, za příkazem *return* již nemůže následovat funkce *LeaveCriticalSection() a* proto musel být atribut *value* umístěn do návratové proměnné *ret.* Elegantněji zle tuto situaci vyřešit použitím této třídy:

```
class cCriticalSection
{
protected:
        CRITICAL_SECTION *mLock;
public:
        cCriticalSection(CRITICAL_SECTION &lock){
                mLock = &lock;
                EnterCriticalSection(mLock); //Vstup do krit. sekce v konstruktoru

        }
        ~cCriticalSection(){
                LeaveCriticalSection(mLock); //Vystoupení z krit. sekce v destruktoru
        }
};
```

Tuto třídu *cCriticalSection* pak použijeme v naší metodě takto:

```
int Trida::GetValue()
{
        cCriticalSection criticalSection(kritickaSekceTridy);
        return value;
}
```

Jakmile je vytvořena instance třídy *cCriticalSection, v* jejím konstruktoru dojde ke vstoupení do kritické sekce. Když je metoda *GetValue() u* konce, je automaticky zavolán destruktor třídy *cCriticalSection, ve* kterém je kritická sekce opuštěna. Tento postup je použit v celé implementaci paralelismu.

3.1.2 Mutex

Princip práce s mechanismem *mutex* je podobný jako s kritickými sekcemi. Celá procedura vypadá následovně:

```
HANDLE mMutex;  //definice proměnné
mMutex = CreateMutex(NULL, false, NULL); //inicializace mutexu
WaitForSingleObject(mMutex, INFINITE); //zamknutí mutexu s nastaveným timeout na INFINITE
/*kód ve kterém by nmohlo dojít ke kolizi více vláken*/
ReleaseMutex(mMutex); //uvolnění mutexu
```

Funkce *WaitForSingleObject()* má dva argumenty. Prvním je proměnná typu *HANDLE* která představuje náš *mutex*. Druhým je nastavení doby čekání (timeout) na uvolnění *mutexu* V našem případě je nastavena na *INFINITE,* což ve výsledku znamená čekání do doby uvolnění *mutexu* metodou *ReleaseMutex()*, neboť nechceme riskovat situaci dvou vláken v jedné chráněné sekci.

3.1.3 Shared Mutex

Tento synchronizační mechanismus je součástí knihovny Boost vyvinuté pro jazyk C++. Jeho výhodou je, že jej není nutné inicializovat jako ostatní synchronizační mechanismy a umožňuje jak výlučné, tak sdílené uzamykání.

Použití vypadá následovně:

```
typedef boost::shared_mutex Lock;                    //definice typu zámku
typedef boost::unique_lock< Lock > WriteLock;    //definice výlučného zámku
typedef boost::shared_lock< Lock > ReadLock;     //definice sdíleného zámku

Lock mLock;
{
        WriteLock writeLock(mLock); //Zamknutí výlučným zámkem
        /*kód ve kterém by mohlo dojít ke kolizi více vláken*/
}       //zde dojde k automatickému uvolnění zámku

{
        ReadLock readLock(mLock); //Zamknutí sdíleným zámkem
        /*zde se může nacháze libovolný počet vláken není-li už držen zámek WriteLock*/
}       //zde dojde k automatickému uvolnění zámku
```

Uvolnění proměnné proběhne automaticky v případech, kdy se volá destruktor třídy, což je v tomto konkrétním příkladě při výstupu z úseku definovaného složenými závorkami.

3.1.4 Metered Section

Práce s *Metered Section* je následující:

```
LPMETERED_SECTION mMetered; //definice proměnné
mMetered = CreateMeteredSection(1,INFINITE,NULL); //inicializace metered section
EnterMeteredSection(mSection, 10); //vstoupení do chráněné sekce s nastavením timeout
/*kód ve kterém by nmohlo dojít ke kolizi více vláken*/
LeaveMeteredSection(mSection, 1, NULL); //vystoupení z chráněné sekce
CloseMeteredSection(mMetered); //definice proměnné
```

Funkce EnterMeteredSection(mSection, 10) má opět možnost, stejně jako u *Mutexu*, nastavit dobu čekání na uvolnění sekce. Pokud je nastavena např. na 10, pak se po 10ms provede vstup do sekce i přesto, že nedošlo k jejímu uvolnění.

3.2 Vytvoření jednotlivých vláken

K vytvoření vláken bylo použito knihovny OpenMP. Tato knihovna je standardně obsažena v knihovnách dodávaných s produktem Microsoft Visual C++, její použití je jednoduché a má řadu výhod.

Spuštění více vláken pro provedení dotazů je s použitím OpenMP provedeno následovně:

```
omp_set_num_threads(2); //nastavení počtu vláken které má OpenMP vytvořit na 2
tmrQuery.Start();       //odstartování časomíry pro změření délky trvání dotazu
#pragma omp parallel for //spuštění následujícího cyklu FOR v nastaveném počtu vláken
for (int i = 0 ; i < count ; i++) //"i" představuje jednotlivé dotazy v jejich
{                                 //předgenerované kolekci
        cItemStream* resultSet; //objekt reprezentující výsledky dotazu
        resultSet = Tree->RangeQuery(
                                qlCollection[i], //dolní mez dotazu
                                qhCollection[i], //dolní mez dotazu
                                &config,        //nastavení pro dotaz
                                NULL);
       resultSet->CloseResultSet(); //uložení výsledku dotazů
}
tmrQuery.Stop();  //zastavení časomíry
```

Příkazem *#pragma omp parallel for* dojde k vykonání následujícího cyklu *for* v počtu vláken definovaném ve funkci *omp_set_num_threads()*, přičemž proběhne automaticky rovnoměrné rozdělení práce pro jednotlivá vytvořená vlákna.

Jedna z velkých výhod tohoto přístupu je možnost kdekoli v kódu zjistit funkcí *omp_get_thread_num(),* které vlákno momentálně kód zpracovává a odpadává tímto nutnost předávat číslo vlákna do jednotlivých volaných funkcí nebo metod tříd.

3.3 Implementace synchronizačních mechanismů

Nejprve je třeba vložit prvky do R-stromu z předvytvořené kolekce prvků. Zjednodušený postup vypadá následovně:

```
cRTree<tKey> *Tree = new cRTree<tKey>(); //definice třídy reprezentující R-strom
cQuickDB *quickDB = new cQuickDB();     //kořenová třída celé hierarchie



for (int i = 0 ; i < COUNT ; i++)
{
        //vkládání prvků do R-stromu z předvytvořené kolekce
        Tree->Insert(dataCollection[i], data);
}
```

Máme R-strom s vloženými přegenerovanými prvky a dotazy. Dotazy probíhají použitím následující konstrukce:

```
for (int i = 0 ; i < count ; i++) //"i" představuje jednotlivé dotazy v jejich
{                                 //předgenerované kolekci
        cItemStream* resultSet; //objekt reprezentující výsledky dotazu
        resultSet = Tree->RangeQuery(
                                qlCollection[i], //dolní mez dotazu
                                qhCollection[i], //dolní mez dotazu
                                &config,        //nastavení pro dotaz
                                NULL);
       resultSet->CloseResultSet(); //uložení výsledku dotazů
}
```

Kde proměnná *Tree* představuje *R-strom,* který je již vytvořen. Vstoupíme-li do definice metody *RangeQuery()*, nachází se hned na prvním řádku následující:

```
cItemStream* resultSet = mQuickDB->GetResultSet();
```

Kde proměnná *resultSet* typu *cItemStream* představuje objekt s nalezenými výsledky

dotazovaného rozsahu. Důležité zde je, že je získána z instance třídy *cQuickDB*, která v tomto konkrétním případě provádění rozsahových dotazů slouží k ukládání výsledku dotazů a musí být sdílená pro všechny běžící vlákna. Jedná se tedy o první třídu, kterou je třeba ošetřit synchronizačními mechanismy.

Zde je ukázka její metody *GetResultSet()* která je ošetřena proti chybám souběhu více vláken:

```
cItemStream* cQuickDB::GetResultSet()
{
        //vstup do kritické sekce a její opuštění destruktorem
        cCriticalSection criticalSection(mCriticalSection);
        return mResultSet[mResultSetPointer--];
}
```

Konkrétně se jedná o použití mechanismu kritické sekce. Stejně jako v tomto případě byly ošetřeny její ostatní metody.

Prováděním dotazu na prvky R-stromu dochází k jeho prohledávání. V QuickDB je to řešeno tak, že je pro jednotlivé uzly R-stromu vytvořena třída *cNodeCache* představující *cache* pro jednotlivé uzly. Každý uzel R-stromu má nastaven svůj vlastní index reprezentovaný celým číslem. Je-li např. potřeba získat konkrétní uzel, zavolá se na instanci třídy *cNodeCache*, ve které jsou uzly uloženy, metoda *Read() s* argumentem indexu požadovaného uzlu a v návratovém typu je tento uzel vrácen.

Zde-je rozdíl oproti běžnému přístupu pro implementaci stromových datových struktur, kdy jsou data stromu uložena v paměti a o jejich správu (inicializaci a destrukci objektu) je postaráno automaticky. Používání této *cache* je z důvodu rychlejší práce s pamětí a zabránění vzniku její fragmentace, ale nevýhodou je, že musí být opět sdílena mezi paralelně běžícími vlákny, kvůli čemuž je potřebné ošetřit jí také synchronizačními mechanismy. Navíc v případě čtení z této *cache* se nejedná pouze o operace čtení, ale jsou prováděny i operace zápisu, např. modifikace různých statistik. Proto není možné mnoho jejich metod uzamykat pouze sdíleným zámkem (pouze v případě synch. mechanismu *Shared Mutex*, který sdílený zámek umožňuje).

Třídy *cNodeCache* a *cQuickDB* jsou v případě dotazů jediné, které je nutné ošetřit synchronizačními mechanismy. V případě implementace paralelismu i pro vkládání prvků bychom museli řešit i zamykání jednotlivých uzlů a větví a to i pro dotazování, neboť při provádění paralelního dotazování a zároveň vkládání by mohlo docházet ke kolizi. Jelikož však uvažujeme pouze dotazy, stačí ošetřit jednotlivé třídy proti chybám souběhu.

4 Testování

4.1 Rozsahové dotazování

Testování dotazů v QuickDB bylo provedeno pro srovnání rychlosti sekvenčního a paralelního přístupu. Pro prvky v R-stromu bylo použito před-generovaných kolekcí. Každá kolekce obsahuje určitý počet prvků a dimenzí. Přehled kolekcí pro vkládání prvků ukazuje následující tabulka:

Jméno kolekce	Počet prvků	Počet dimenzí	Velikost [MB]
Docword	483 450 157	3	8 100
Meteo	57 852 305	5	1 176
Poker	1 000 000	11	23
Tiger	5 889 786	2	115
Xml	17 646 635	9	431

Tabulka 6: Použité kolekce prvků

Pro dotazování byly použity také před-generované kolekce, avšak s dotazy. Vygenerování těchto kolekcí dotazů bylo provedeno pro jednotlivé kolekce prvků (tj. zvlášť pro kolekci Docword, Meteo atd.). Celkem jsou pro každou kolekci prvků vygenerovány tři kolekce dotazů, lišící se intervalem počtu výsledných prvků pro každý typ dotazu. Tyto kolekce jsou shrnuty v následující tabulce:

Označení	Počet výsledných prvků dotazu	Počet dotazů
Bodové dotazy	<1,1>	1000
Dotazy středního pokrytí	<2, 1000>	1000
Dotazy vysokého pokrytí	<1000, 100000>	1000

Tabulka 7: Použité typy dotazování pro všechny kolekce

Dotazování bylo dále prováděno na několika typech synchronizačních mechanizmů, aby bylo možné porovnat jejich výkon. Jsou shrnuty v následující tabulce:

Označení	Název knihovny (pro jazyk c++)	Popis
Critical Section	windows.h	prvek knihovny Windows API
Mutex	windows.h	prvek knihovny Windows API
Shared Mutex	boost/thread/sync/sharedmutex.h	prvek knihovny Boost
Metered Section	meteredsection.h	dodaný vedoucím bak. práce

Tabulka 8: Použité typy synchronizačních mechanismů

Celé testování proběhlo na serverovém počítači naší školy s těmito parametry:

Jméno počítače	Dbedu
Operační systém	Windows Server 2008 R2 Datacenter
Typ systému	64-bitový operační systém
Procesor	2x Intel Xeon X5670 2,93Ghz
Počet jader procesoru	2x6 jader / 2x12 vláken
Paměť RAM	96,0 GB

Tabulka 9: Parametry testovacího počítače

Protože byly použity před-generované kolekce vkládaných prvků, v následující tabulce budou ukázány charakteristiky jednotlivých vytvořených R-stromů pro tyto kolekce:.

Jméno kolekce	Výška stromu	Počet vnitřních položek	Počet listových položek	Počet vnitřních uzlů	Počet listových uzlů	Dimenzí
Docword	4	13 700 008	483 450 156	45 023	7 250 409	3
Meteo	4	2 054 235	57 852 304	10 098	1 107 879	5
Poker	3	71 365	1 000 000	869	35 059	11
Tiger	3	88 844	5 889 786	208	51 791	2
Xml	4	1 212 307	17 646 635	10 926	630 333	9

Tabulka 10: Charakteristika R-stromů vygenerovaných nad jednotlivými kolekcemi

Protože testovací počítač Dbedu slouží zároveň jako server na VŠB-TUO a má k němu přístup více uživatelů najednou, vyplývá z toho možnost chyby měření kvůli jeho možných odlišných stavech při probíhaném měření. Tento fakt je třeba brát v úvahu při vyvozování závěrů z naměřených hodnot.

Následují výsledky testů pro jednotlivé kolekce prvků. Jak už bylo řečeno, jsou pro každou kolekci prvků provedeny tři typy dotazů.

4.1.1 Výsledky testů

Zamykací mechanismus	Typ dotazů	Čas zpracování dotazů v jednotlivých vláknech [s]							
		sekv.	2	3	4	5	10	15	20
Critical section	Bodové	0,028	0,025	0,022	0,023	0,022	0,025	0,028	0,030
	Střední pokrytí	1,1	1,0	0,9	0,9	0,9	0,8	1,1	1,2
	Vysoké pokrytí	36,0	43,8	18,8	48,5	48,5	52,1	53,0	55,0
Mutex	Bodové	0,028	0,058	0,075	0,080	0,030	0,091	0,117	0,098
	Střední pokrytí	1,1	1,3	1,9	2,0	2,0	2,1	2,1	2,1
	Vysoké pokrytí	36,0	40,4	26,9	48,5	49,1	52,1	34,1	52,5
Shared Mutex	Bodové	0,028	0,026	0,024	0,024	0,024	0,024	0,033	0,039
	Střední pokrytí	1,1	1,0	0,9	0,9	0,9	0,8	1,1	1,2
	Vysoké pokrytí	36,0	32,4	16,7	33,2	33,8	43,4	25,3	55,1
Metered Section	Bodové	0,028	0,026	0,024	0,025	0,024	0,037	0,000	0,034
	Střední pokrytí	1,1	1,0	0,9	0,9	0,9	0,9	1,1	1,2
	Vysoké pokrytí	36,0	33,5	17,3	39,8	40,5	42,5	22,0	43,0

Tabulka 11: Výsledky testů pro kolekci Docword

Zamykací mechanismus	Typ dotazů	Čas zpracování dotazů v jednotlivých vláknech [s]							
		sekv.	2	3	4	5	10	15	20
Critical section	Bodové	0,046	0,042	0,039	0,038	0,038	0,043	0,048	0,048
	Střední pokrytí	0,09	0,08	0,07	0,07	0,07	0,07	0,10	0,11
	Vysoké pokrytí	21,3	17,2	14,7	13,8	13,1	13,0	14,0	14,13
Mutex	Bodové	0,046	0,131	0,178	0,180	0,170	0,194	0,206	0,199
	Střední pokrytí	0,09	0,19	0,23	0,25	0,25	0,27	0,27	0,28
	Vysoké pokrytí	21,3	19,8	18,0	18,7	18,3	21,1	22,9	24,1
Shared Mutex	Bodové	0,046	0,055	0,049	0,046	0,043	0,045	0,061	0,069
	Střední pokrytí	0,09	0,09	0,08	0,08	0,08	0,07	0,12	0,12
	Vysoké pokrytí	21,3	17,5	15,1	14,5	13,9	16,1	20,0	23,0
Metered Section	Bodové	0,046	0,053	0,055	0,062	0,071	0,099	0,105	0,110
	Střední pokrytí	0,09	0,09	0,09	0,09	0,10	0,14	0,15	0,17
	Vysoké pokrytí	21,3	17,7	15,3	15,4	14,3	15,4	16,5	17,1

Tabulka 12: Výsledky testů pro kolekci Meteo

Zamykací mechanismus	Typ dotazů	Čas zpracování dotazů v jednotlivých vláknech [s]							
		sekv.	2	3	4	5	10	15	20
Critical section	Bodové	0,053	0,055	0,050	0,050	0,048	0,054	0,058	0,060
	Střední pokrytí	2,0	1,9	1,6	1,5	1,5	1,5	1,5	1,5
	Vysoké pokrytí	18,9	13,8	12,0	11,4	10,9	10,4	15,5	15,7
Mutex	Bodové	0,053	0,264	0,279	0,288	0,286	0,301	0,305	0,310
	Střední pokrytí	2,0	5,3	6,7	7,0	6,8	7,1	7,2	7,2
	Vysoké pokrytí	18,9	26,2	37,0	40,0	39,7	41,0	42,0	42,0
Shared Mutex	Bodové	0,053	0,078	0,063	0,060	0,055	0,060	0,076	0,076
	Střední pokrytí	2,0	2,3	1,9	2,7	1,6	1,6	2,1	2,2
	Vysoké pokrytí	18,9	15,7	13,2	12,1	11,4	10,9	16,2	16,1
Metered Section	Bodové	0,053	0,160	0,151	0,151	0,147	0,157	0,000	0,320
	Střední pokrytí	1,97	1,61	1,43	1,37	1,33	1,48	2,07	2,93
	Vysoké pokrytí	18,9	20,7	19,2	18,3	18,2	18,7	27,0	26,3

Tabulka 13: Výsledky testů pro kolekci Poker

Zamykací mechanismus	Typ dotazů	Čas zpracování dotazů v jednotlivých vláknech [s]							
		sekv.	2	3	4	5	10	15	20
Critical section	Bodové	0,020	0,025	0,023	0,023	0,024	0,026	0,038	0,042
	Střední pokrytí	0,024	0,030	0,030	0,030	0,030	0,030	0,046	0,050
	Vysoké pokrytí	4,4	3,2	2,4	2,2	2,1	1,9	2,7	2,7
Mutex	Bodové	0,020	0,057	0,075	0,079	0,079	0,085	0,093	0,095
	Střední pokrytí	0,024	0,062	0,082	0,085	0,085	0,095	0,108	0,106
	Vysoké pokrytí	4,4	2,9	2,5	2,6	2,6	3,0	3,1	3,1
Shared Mutex	Bodové	0,020	0,027	0,026	0,025	0,026	0,025	0,037	0,000
	Střední pokrytí	0,024	0,033	0,030	0,030	0,030	0,033	0,045	0,052
	Vysoké pokrytí	4,4	3,3	2,5	2,2	2,1	2,0	2,6	2,7
Metered Section	Bodové	0,020	0,020	0,018	0,018	0,019	0,019	0,021	0,042
	Střední pokrytí	0,024	0,023	0,022	0,022	0,022	0,023	0,064	0,038
	Vysoké pokrytí	4,4	2,5	2,1	1,8	1,6	1,4	2,0	2,0

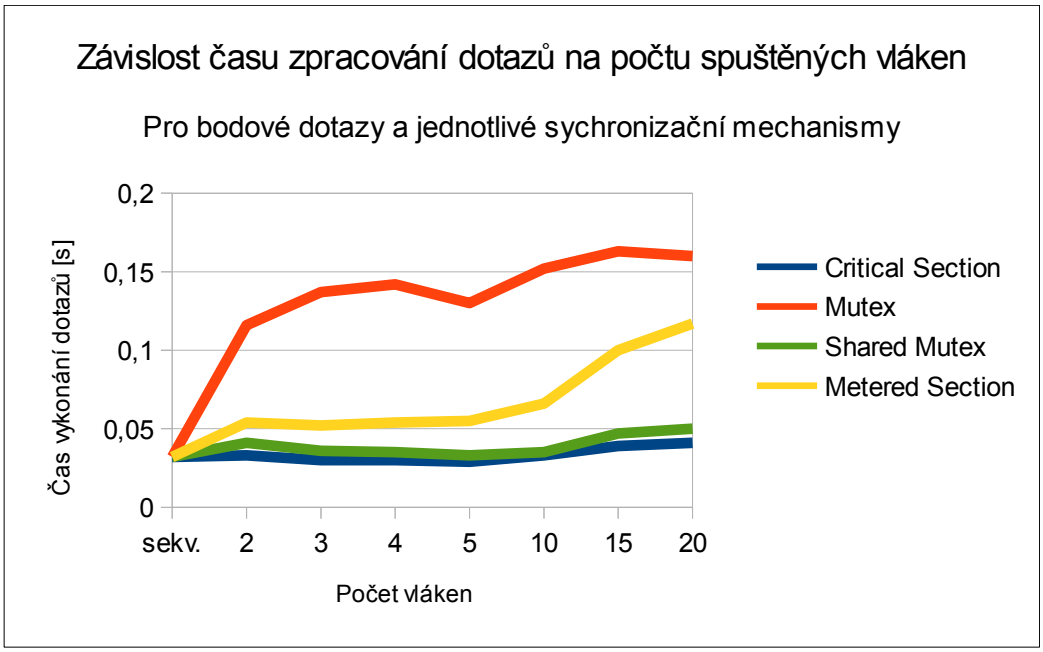
Tabulka 14: Výsledky testů pro kolekci Tiger

Zamykací mechanismus	Typ dotazů	Čas zpracování dotazů v jednotlivých vláknech [s]							
		sekv.	2	3	4	5	10	15	20
Critical section	Bodové	0,013	0,016	0,015	0,015	0,015	0,017	0,021	0,025
	Střední pokrytí	0,032	0,038	0,037	0,037	0,037	0,045	0,059	0,064
	Vysoké pokrytí	17,6	10,9	13,5	10,1	13,2	11,0	14,2	15,0
Mutex	Bodové	0,013	0,069	0,078	0,083	0,083	0,090	0,093	0,096
	Střední pokrytí	0,032	0,153	0,160	0,164	0,176	0,185	0,204	0,200
	Vysoké pokrytí	17,6	13,7	19,2	16,5	20,4	18,8	23,7	24,0
Shared Mutex	Bodové	0,013	0,020	0,019	0,018	0,018	0,021	0,027	0,000
	Střední pokrytí	0,032	0,042	0,038	0,038	0,038	0,045	0,057	0,071
	Vysoké pokrytí	17,6	10,9	13,4	9,8	15,2	16,5	27,5	29,5
Metered Section	Bodové	0,013	0,013	0,012	0,012	0,013	0,017	1,012	0,080
	Střední pokrytí	0,032	0,031	0,030	0,000	0,031	0,041	0,075	0,084
	Vysoké pokrytí	17,6	11,1	14,1	10,3	13,9	13,7	17,2	17,6

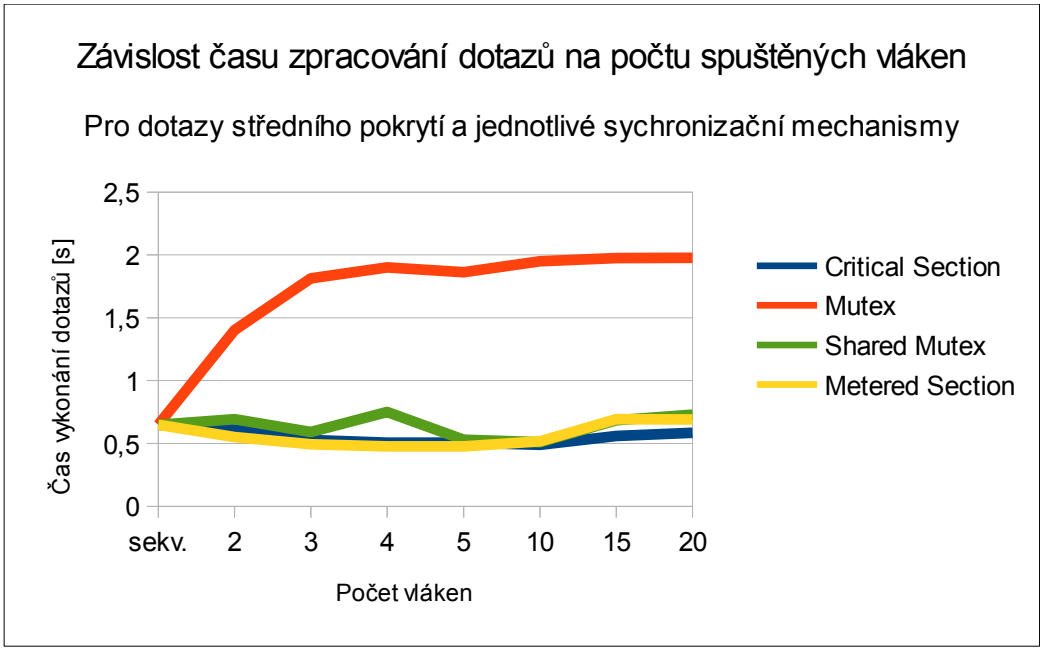
Tabulka 15: Výsledky testů pro kolekci Xml

4.1.2 Výsledky testů v grafech

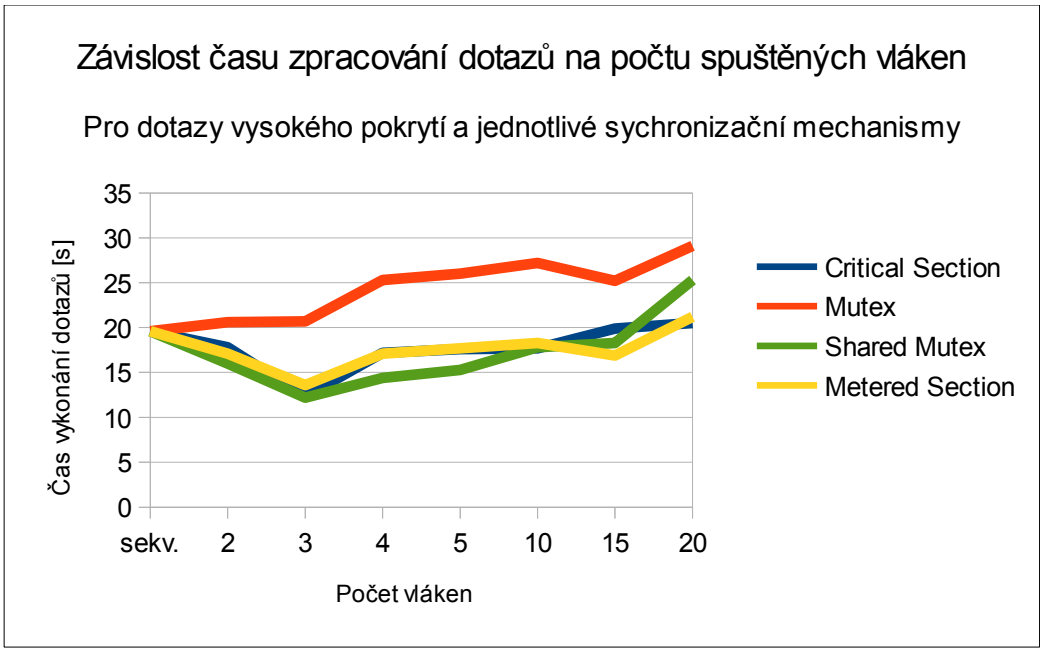
Pro výsledky testů byl proveden aritmetický průměr časových hodnot naměřených pro jednotlivé typy dotazů a pro jednotlivé uzamykací mechanismy a to nad všemi kolekcemi prvků dohromady. Následně vše bylo zaznamenáno do grafů.



Ilustrace 5: Závislost času zpracování dotazů na počtu spuštěných vláken pro bodové dotazy



Ilustrace 6: Závislost času zpracování dotazů na počtu spuštěných vláken pro dotazy středního pokrytí



Ilustrace 7: Závislost času zpracování dotazů na počtu spuštěných vláken pro dotazy vysokého pokrytí

5 Závěr

5.1 Zhodnocení dosažených výsledků měření

5.1.1 Výkonost paralelismu

Z výsledků měření je patrné, že výhoda paralelního zpracování dotazů nad implementací R-stromu není taková, jak by se možná očekávalo. Trvá-li zpracování množiny dotazů sekvenčně čas *T*, řekli bychom, že ve dvou vláknech se bude blížit k čase *T/2*, ve třech vláknech k čase *T/3*, ve čtyřech k čase *T/4* atd. a že jsme omezeni jen počtem vláken, které je schopen poskytnout procesor, na kterém aplikace běží.

Hlavní důvody, proč tomu tak v tomto konkrétním případě není, jsou tyto:

- Většina práce vláken nespočívá na náročných výpočtech, které by musel provádět procesor, ale na práci s pamětí, ve které je celý R-strom uložen. Z tohoto důvodu je hlavní brzdou rychlost práce s pamětí.
- V QuickDB je práce s pamětí optimalizována naprogramovanou vlastní *cache*. Veškeré datové objekty, které je potřeba vytvořit, například jednotlivé uzly R-stromu, jsou vytvářeny a spravovány skrze tuto *cache*. Jelikož je však nutno ji sdílet mezi jednotlivými vlákny, je ji potřeba také zamykat. Pokud je zamknutá, ostatní vlákna musí čekat, než na ně přijde řada.
- Je třeba započítat určité režie pro vytváření jednotlivých vláken a pro synchronizační mechanismy, ty totiž v běžném sekvenčním přístupu nejsou. Toto se negativně projeví hlavně při zpracovávání dotazů nízkého pokrytí, což potvrzují výsledky testů.
- Přes veškerou snahu vytvořit v QuickDB dobrou logiku paralelismu je na ní stále co vylepšovat. Dotáhnout paralelismus v této aplikaci do nejoptimálnější podoby mohou nejlépe ti, kteří ji sami naprogramovali a vědí do detailu, jak pracuje.

Nejznatelnější jsou výhody paralelismu při zpracování dotazů vysokého pokrytí. U dotazů nízkého a částečně i středního pokrytí hraje velkou roli režie při vytváření vláken a režie spojená se synchronizačními mechanismy.

Zajímavý je údaj, který byl naměřen u kolekce Docword všemi synchronizačními mechanismy pro dotazy vysokého pokrytí. Ten ukazuje pro tři vlákna dobu trvání až 16 vteřin (viz. tabulka č. 11). Oproti sekvenčnímu přístupu, který trval 36 vteřin je to dvojnásobné zrychlení. Zřejmě je tomu tak proto, že se za jistých podmínek, které v této situaci vznikly (dobrá fáze jednotlivých vláken atd.), vlákna takřka navzájem neblokovala.

5.1.2 Výkonost jednotlivých synchronizačních mechanismů

Při testech jsem kromě porovnávání paralelního přístupu se sekvenčním také porovnával výkon různých synchronizačních mechanizmů. Z výsledků testů si nejlépe vedou mechanismy *Critical Section, Shared Mutex* a *Metered Section*, relativně špatně dopadl *Mutex*.

Je třeba říct, že v určitých situacích by mohl být nejrychlejší *Shared Mutex*, neboť umožňuje

uzamykat i sdílenými zámky, kdežto ostatní mechanismy ne.

Výhodou *Mutex* a *Metered Section* je možnost nastavení doby čekání pro odemčení zámku, což se ale v tomto případě testování nedalo využít.

5.2 Osobní poznatky

Promýšlení možností paralelismu v rámci této bakalářské práce bylo pro mně přínosné a zajímavé. Považuji to za odvětví, ve kterém je možné vymyslet velké množství zajímavých přístupů k vytvoření dobrého paralelismu.

Při této práci jsem si uvědomil pár věcí, které budu v budoucnu používat, budu-li chtít ve své aplikaci pracovat s vlákny. Jedna z nich je, že se velmi vyplatí dodržovat pravidla OOP, zvláště pak vlastnosti zapouzdření. Další důležitý poznatek spočívá v tom, že je dobré vytvořit architekturu aplikace tak, aby bylo mezi sebou sdíleno co nejméně tříd a proměnných, ve kterých by pak docházelo k blokování vláken. Když už je třeba proměnné sdílet, je dobré manipulovat s nimi např. v metodě třídy pokud možno v jednom úseku, aby tak mohla být délka zamykání co nejkratší.

Závěrem je třeba říct, že byly splněny všechny požadavky kladené na tuto bakalářskou práci. Výjimkou bylo jen porovnání s komerčními SŘBD, kde byl tento úkol po domluvě s vedoucím bakalářské práce nahrazen porovnáním více synchronizačních mechanismů mezi sebou.

Přílohy

A Použitá literatura

[1] KOPETSCHKE, Igor. Vlákna. [online]. [cit. 2013-03-15]. Dostupné z: http://www.nti.tul.cz/cz/images/3/31/DPG-3.pdf

[2] KOLÁŘ, Petr. Operační systémy. [online]. [cit. 2013-03-19]. Dostupné z: http://www.nti.tul.cz/~kolar/os/os.pdf

[3] GUTTMAN, Antonin. R-Trees: A Dynamic Index Structure for Spatial Searching. In Proceedings of Annual Meeting. Boston (MA), ACM Press, 1984. s. 47–57. Dostupné z: www.sai.msu.su/megera/postgres/gist/papers/gutman-rtree.pdf.

[4] PROCHÁZKOVÁ, Jana. R-stromy a jejich paralelizace. Ostrava, 2008. 59 s. Diplomová práce. VŠB-TUO FEI.

[5] HARDER, Theo, REUTER, Andreas. Pinciples of Transaction-Oriented Database Recovery. ACM Computing Surveys, 1983, vol. 15, no. 4, s. 287-317

[6] POKORNÝ, Jaroslav, ŽEMLIČKA, Michal. Základy implementace souboru a databází. 2. vyd. Praha, Karolinum, 2004, 211 s. ISBN 80-246-0837-5

B Obsah CD

Na přiloženém CD je adresář *bakalářská práce* který obsahuje:

- adresář *Implementace*
    - adresář *QuickDB* ve kterém jsou třídy z aplikace QuickDB, které byly upraveny
    - adresář *Utils* obsahující třídy, které byly napsány v rámci této bakalářské práce pro účely testování
- adresář *Text* ve kterém je text této bakalářské práce ve formátu pdf